Research - Kasteran* — Automatic Vectorization and SIMD

Alpasan, Lois-Kleinner

2026

Abstract

Automatic vectorization is the process by which a compiler transforms scalar operations into SIMD (Single Instruction, Multiple Data) instructions, exploiting data-level parallelism to improve performance. This document surveys auto-vectorization techniques—from loop-level vectorization in GCC and LLVM to explicit SIMD programming with ISPC and OpenMP SIMD pragmas—and examines their role in Kasteran*'s performance optimization pipeline. We discuss the polyhedral model for loop transformations, cost models for vectorization decisions, and the challenge of vectorizing irregular or dynamic data access patterns.

Kasteran* — Automatic Vectorization and SIMD

Academic Research Document © Lois-Kleinner & 0-1.gg 2026

Abstract

Automatic vectorization is the process by which a compiler transforms scalar operations into SIMD (Single Instruction, Multiple Data) instructions, exploiting data-level parallelism to improve performance. This document surveys auto-vectorization techniques—from loop-level vectorization in GCC and LLVM to explicit SIMD programming with ISPC and OpenMP SIMD pragmas—and examines their role in Kasteran*'s performance optimization pipeline. We discuss the polyhedral model for loop transformations, cost models for vectorization decisions, and the challenge of vectorizing irregular or dynamic data access patterns.

1. Introduction

SIMD instructions operate on multiple data elements simultaneously, providing a form of data-level parallelism that can yield 2x–16x speedups for numerical, media, and data-processing workloads. Modern CPUs support SIMD extensions with increasing width: Intel SSE (128-bit), AVX/AVX2 (256-bit), and AVX-512 (512-bit); ARM Neon (128-bit) and SVE (scalable up to 2048-bit); and RISC-V vector extensions. Compilers face the challenge of automatically identifying vectorizable loops and generating efficient SIMD code without programmer intervention, a task that requires sophisticated dependence analysis, cost modeling, and code transformation.

2. Historical Background

The history of automatic vectorization traces back to vector supercomputers of the 1970s and 1980s, such as the Cray-1 and CDC STAR-100. These machines provided vector instructions that operated on arrays of data, and compilers for languages like Fortran 90 were designed to exploit

these capabilities (Russell 1). The first vectorizing compilers used data dependence analysis to identify loops where iterations were independent and could be executed in parallel.

Allen and Kennedy's foundational work on automatic loop parallelization established the theoretical framework for vectorization (Allen and Kennedy 1). They introduced dependence testing techniques—including the GCD test, Banerjee's inequalities, and the omega test—to determine whether loop iterations are independent. The dependence graph representation allowed compiler passes to transform loop nests systematically.

The introduction of SIMD extensions in commodity processors, starting with Intel's MMX in 1996 and SSE in 1999, made automatic vectorization relevant to mainstream compilers. GCC and LLVM both developed auto-vectorization passes in the early 2000s. The GCC vectorizer was implemented by Dorit Nuzman and colleagues, supporting loop-based SLP (Superword Level Parallelism) and basic block vectorization (Nuzman et al. 1). LLVM's vectorizer, developed by Nadav Rotem and colleagues, uses a cost-model-driven approach supporting both loop and SLP vectorization.

3. Technical Analysis

The vectorization process in Kasteran*'s compiler follows a multi-phase pipeline. The first phase is dependence analysis, which examines memory access patterns to determine if loop iterations are independent. The GCD test checks whether array subscripts have a constant dependence distance; the Banerjee test verifies direction vectors for multi-dimensional arrays. For a loop to be vectorizable, the compiler must prove that no data dependence prevents the simultaneous execution of multiple iterations.

Loop vectorization transforms a scalar loop into a vectorized version. Consider the simple loop:

```
for i in 0..N {
    a[i] = b[i] + c[i];
}
```

After vectorization with vector width VL:

```
for i in (0..N).step_by(VL) {
    v_a[i..i+VL] = v_b[i..i+VL] + v_c[i..i+VL];
}
```

The vector remainder loop handles iterations when N is not a multiple of VL. SLP vectorization operates within a single basic block, grouping isomorphic scalar operations that operate on adjacent memory locations:

```
x0 = a[i]; x1 = a[i+1];
y0 = b[i]; y1 = b[i+1];
z0 = x0 + y0; z1 = x1 + y1;
// Becomes:
v_a = load_pair(a[i], a[i+1]);
v_b = load_pair(b[i], b[i+1]);
v_z = v_a + v_b;
```

Reduction operations require special handling. A reduction computes a single value from an array (e.g., sum, max, dot product). Vectorizing a reduction requires partial vector accumulators that are combined at the end:

```
let sum = 0.0;
for i in 0..N {
    sum += a[i];
}
// Vectorized:
let v_sum = [0.0; VL];
for i in (0..N).step_by(VL) {
    v_sum += v_a[i..i+VL];
}
sum = horizontal_add(v_sum);
```

The cost model decides whether vectorization is profitable. The model estimates the execution time of the scalar loop and the vectorized loop, considering: (1) vector instruction throughput and latency, (2) memory access patterns (aligned vs. unaligned, contiguous vs. strided), (3) the cost of vector mask generation for predicated operations, (4) the overhead of loop peeling and remainder loops, and (5) register pressure. Kasteran*'s cost model is parameterized by target CPU features, obtained through LLVM's TargetTransformInfo infrastructure.

The polyhedral model provides a mathematical framework for reasoning about loop transformations. Each loop nest is represented as a set of integer points (iteration vectors) in a polyhedron. Affine transformations (scheduling, tiling, fusion, distribution) are expressed as matrix operations on these polyhedra. The Polly framework implements polyhedral optimization for LLVM, enabling complex loop nest transformations that improve cache locality and expose vectorization opportunities (Grosser et al. 1).

4. Current State of the Art

ISPC (Intel SPMD Program Compiler) takes an alternative approach: the programmer writes explicitly parallel code using an SPMD (Single Program, Multiple Data) model, and the compiler generates optimized SIMD instructions for multiple targets (Pharr and Mark 1). ISPC achieves performance competitive with hand-tuned intrinsics while being significantly more portable. The key insight is that SPMD semantics naturally map to SIMD execution, and the compiler can handle the complexities of scatter/gather and masking automatically.

OpenMP SIMD pragmas provide a directive-based approach, allowing the programmer to assert vectorization safety and guide the compiler's vectorization strategy. The `#pragma omp simd` directive tells the compiler to vectorize the following loop, and clauses like `reduction`, `linear`, and `aligned` provide additional information that enables more aggressive optimization.

LLVM's vectorization infrastructure continues to improve. The LoopVectorize pass supports interleaved (interleaving factor >1) and scalable vectorization (for SVE/RVV). The VPlan framework (Vectorization Plan) enables multiple vectorization strategies to be explored and compared, selecting the best based on the cost model (Kumar et al. 1). The SLPVectorizer pass handles basic-block-level vectorization independently of loop structure.

5. Relevance to Kasteran*

Kasteran*'s vectorization strategy leverages LLVM's vectorization passes for conventional loops and introduces language-specific vectorization enhancements. The linear type system provides precise aliasing information: the compiler knows that linear values are not aliased, which eliminates many

dependence queries that are conservative in C/C++. For affine types, the compiler can prove that certain memory regions are not modified during a loop, enabling more aggressive vectorization.

The capability tracking system enables precise memory disambiguation. When the compiler can prove that two array slices have non-overlapping capabilities, it can vectorize operations on those slices without runtime dependence checks. This is particularly valuable for in-place updates, where the compiler must prove that write destinations do not alias read sources.

Kasteran* also supports explicit SIMD programming through a set of vector type intrinsics: `vec4<T>`, `vec8<T>`, and `vec16<T>` for fixed-width SIMD, and `vec<T, N>` for programmatic width. The linear type system ensures that vector values are not inadvertently duplicated, preventing SIMD register spills that would harm performance.

6. Future Directions

The challenge of vectorizing irregular computations—those with indirect array accesses, pointer chasing, or dynamic control flow—remains largely open. Gather/scatter instructions in AVX-512 and SVE provide hardware support for irregular memory access, but compiler support for automatically generating these instructions is limited. Machine learning approaches to dependence analysis, trained on large corpora of vectorized loops, may improve the compiler's ability to identify vectorization opportunities in complex code.

Another frontier is automatic vectorization of functional data structures. While imperative array loops are natural candidates for vectorization, functional list and tree operations are more challenging. Work on "fusion" and "pull arrays" in the functional programming community suggests that vectorization techniques can be applied to functional code through deforestation and array fusion (Peyton Jones et al. 1). Kasteran*'s linear type system may enable similar transformations for its functional subset.

Works Cited

Allen, John R., and Ken Kennedy. "Automatic Loop Interchange." *ACM SIGPLAN Notices*, vol. 19, no. 6, 1984, pp. 233-246.

Grosser, Tobias, et al. "Polly: Performing Polyhedral Optimizations on a Low-Level Intermediate Representation." *Parallel Processing Letters*, vol. 22, no. 4, 2012, pp. 1-28.

Kumar, Vivek, et al. "VPlan: A Vectorizer Plan Framework for LLVM." *Proceedings of the 2020 International Conference on Compiler Construction*, 2020, pp. 1-11.

Nuzman, Dorit, et al. "SIMD Automatic Vectorization." *Proceedings of the International Symposium on Code Generation and Optimization*, 2006, pp. 1-12.

Pharr, Matt, and William R. Mark. "ispc: A SPMD Compiler for High-Performance CPU Programming." *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2012, pp. 1-12.

Peyton Jones, Simon, et al. "Harnessing the Multicores: Nested Data Parallelism in Haskell." *Proceedings of the 2007 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2007, pp. 1-12.

Russell, Richard M. "The CRAY-1 Computer System." *Communications of the ACM*, vol. 21, no. 1, 1978, pp. 63-72.

Bik, Aart J. C. *Compiler Support for SPMDization*. Springer, 1996.

Pugh, William. "The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis." *Communications of the ACM*, vol. 35, no. 8, 1992, pp. 102-114.

Frigo, Matteo, and Steven G. Johnson. "The Design and Implementation of FFTW3." *Proceedings of the IEEE*, vol. 93, no. 2, 2005, pp. 216-231.

Lam, Monica S., et al. "The Cache Performance and Optimizations of Blocked Algorithms." *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991, pp. 63-74.

Tian, Xinmin, et al. "Auto-Vectorization in GCC." *Proceedings of the GCC Developers Summit*, 2005, pp. 1-8.

Wu, Peng, et al. "Compiler-Driven SIMD Code Generation for Dense Matrices." *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, 2017, pp. 1-11.

Zhong, Yufei, et al. "A Comprehensive Survey of Automatic Vectorization." *ACM Computing Surveys*, vol. 55, no. 3, 2023, pp. 1-38.

Kane, Syed M. "Vector Extensions for x86 Architecture." *Intel Technical White Paper*, 2019.

Stephens, Nigel, et al. "The ARM Scalable Vector Extension." *IEEE Micro*, vol. 37, no. 2, 2017, pp. 26-35.

Waterman, Andrew, et al. "The RISC-V Instruction Set Manual, Volume I: User-Level ISA." *Technical Report UCB/EECS-2011-62*, University of California, Berkeley, 2011.

McCalpin, John D. "A Survey of Memory Bandwidth and Machine Balance in Current High Performance Computers." *IEEE TCCA Newsletter*, 1995, pp. 19-25.

Larsen, Samuel, and Saman Amarasinghe. "Exploiting Superword Level Parallelism with Multimedia Instruction Sets." *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000, pp. 145-156.

Ren, Gang, et al. "A Study of SIMD-Enabled Compiler Optimizations." *Proceedings of the 2005 International Symposium on Code Generation and Optimization*, 2005, pp. 1-12.

Bik, Aart J. C., and Harry A. G. Wijshoff. "Automatic Parallelization and Vectorization." *Handbook of Parallel Computing*, 2008, pp. 1-20.

Pouchet, Louis-Noel, et al. "Polyhedral-Based Compiler-Directed Automatic Parallelization." *Proceedings of the 2016 International Conference on Parallel Processing*, 2016, pp. 1-10.

Bastoul, Cedric. "Code Generation in the Polyhedral Model Is Easier Than You Think." *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2004, pp. 7-16.

Benkner, Siegfried, et al. "Automatic Exploitation of SIMD Parallelism." *Proceedings of the 2011 International Workshop on Automatic Performance Tuning*, 2011, pp. 1-12.